

Sincronizzazione dei processi e dei thread

Processi concorrenti

P1 e P2 si dicono concorrenti se (alcune) operazioni di P1 sono temporalmente comprese tra la prima operazione di P2 e l'ultima operazione di P2 (o viceversa)

I processi concorrenti sono caratterizzati dal fatto che la loro esecuzione si sovrappone nel tempo.

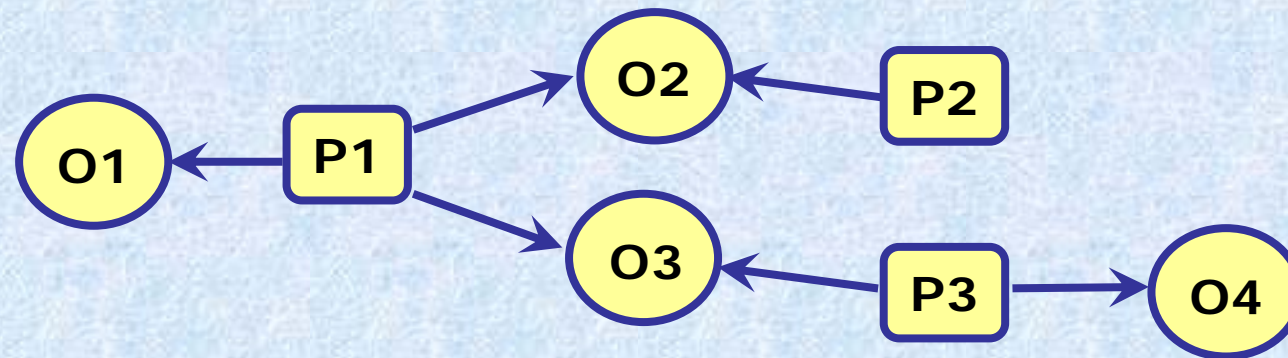
==> contemporaneità reale solo nel caso dei multiprocessori

Ambiente globale/locale

Il sistema è visto come un insieme di processi concorrenti e di oggetti (dati del processo, strutture dati che rappresentano le risorse)

Dal punto di vista della proprietà delle **risorse di memoria**, sono possibili i due modelli ad *ambiente globale* e ad *ambiente locale*

Modello ad ambiente globale

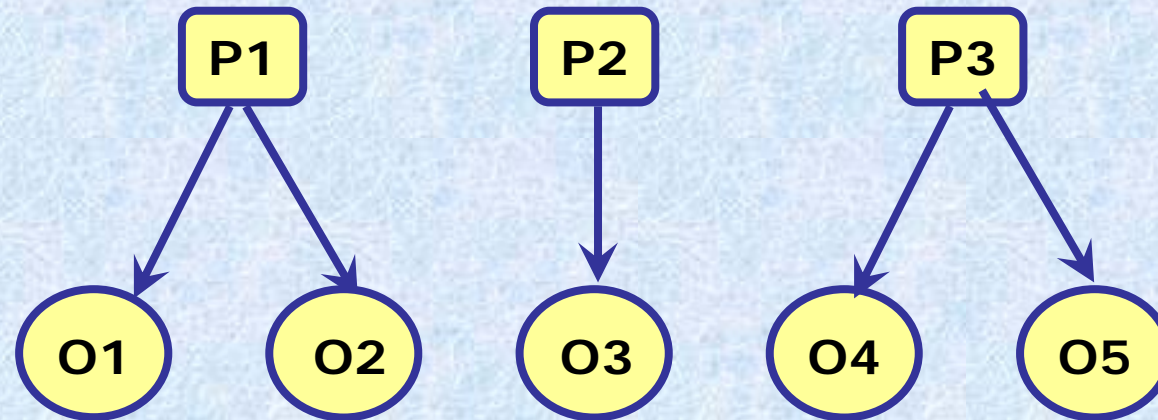


O1, O4 oggetti privati

O2, O3 oggetti comuni

==> Competizione, cooperazione

Modello ad ambiente locale

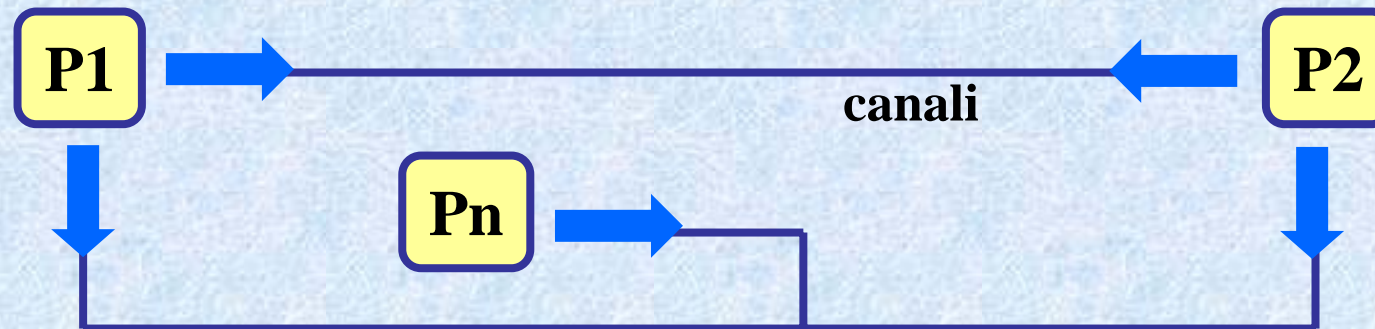


O1, O2, O3, O4, O5 risorse private

==> Competizione attraverso i processi serventi; cooperazione

Ambiente locale

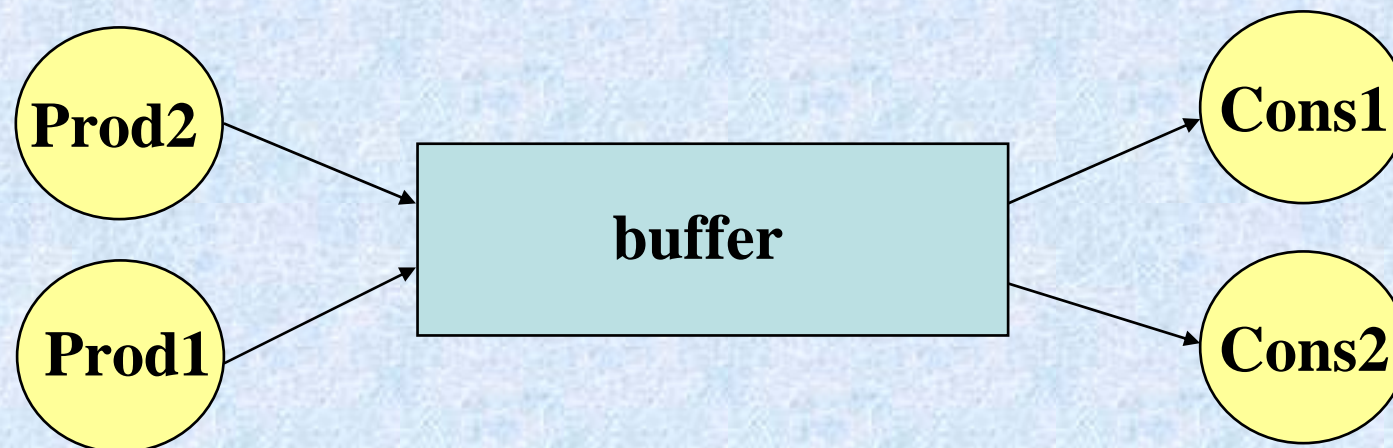
- Ambiente locale (sistemi distribuiti, ma anche sistemi monoprocessore)
- L'ambiente di un processo non è accessibile direttamente da nessun altro processo
 - Ogni forma di cooperazione tra processi (comunicazione, sincronizzazione) avviene tramite lo scambio di messaggi



Processi indipendenti/interagenti

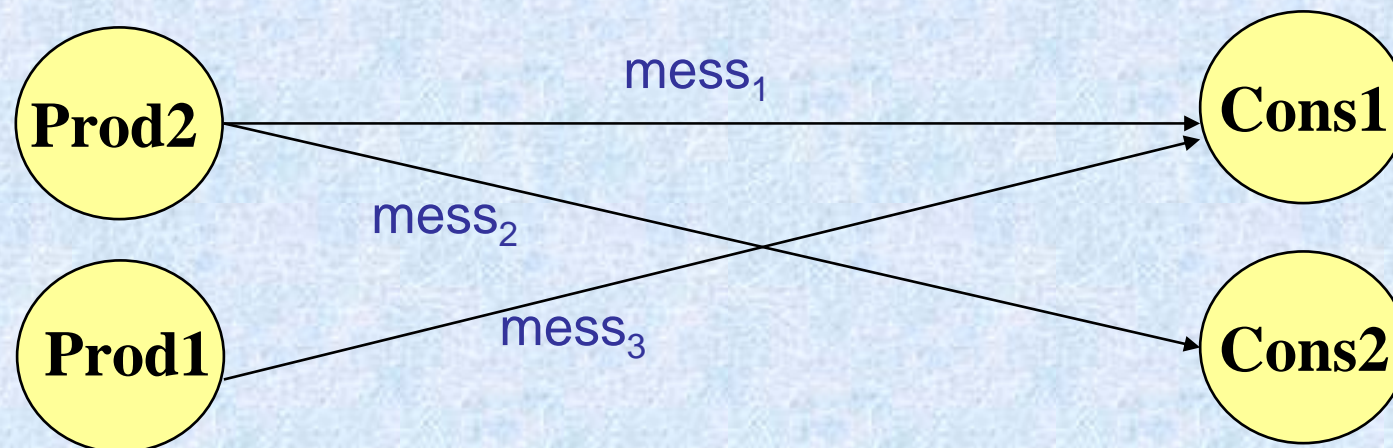
- **Processi indipendenti:**
 - due processi P1 e P2 sono indipendenti se l'esecuzione di P1 non è influenzata da P2, e viceversa
 - Interazioni: solo competizione
 - Comportamento riproducibile
- **Processi interagenti:**
 - due processi P1 e P2 sono interagenti se l'esecuzione di P1 è influenzata da P2, e viceversa
 - Interazioni: competizione e cooperazione
 - Effetto della cooperazione dipende dalla velocità relativa dei processi
 - Comportamento non riproducibile

Processi interagenti



Ambiente globale

Processi interagenti



Ambiente locale

Sincronizzazione tra processi in ambiente globale

Per un corretto funzionamento, è necessario imporre
vincoli ai processi per l'esecuzione delle loro operazioni

Vincoli

- **Competizione:** in generale un solo processo alla volta deve avere accesso alla risorsa comune (mutua esclusione)

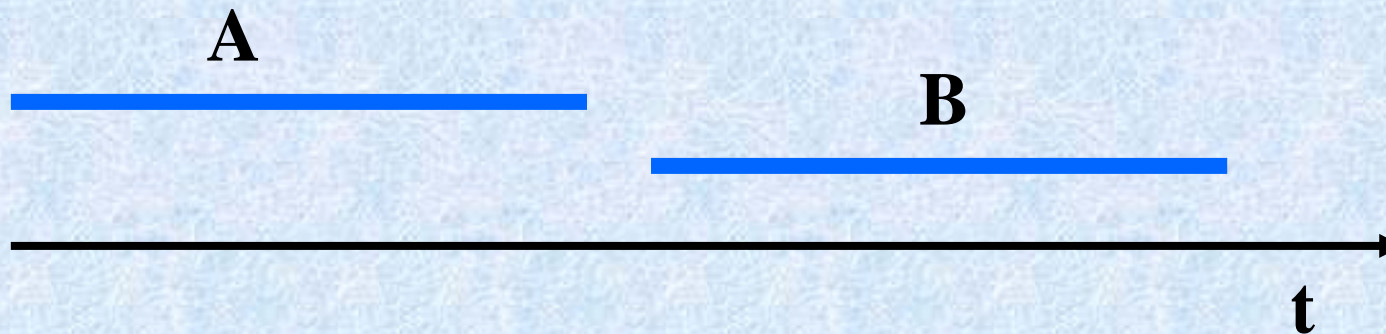
sincronizzazione indiretta o implicita

- **Cooperazione:** le operazioni dei processi cooperanti (esempio: quelle con le quali produttore e consumatore operano sul buffer) devono seguire una sequenza corretta

sincronizzazione diretta o esplicita

Sincronizzazione tra processi in ambiente globale: Problema della mutua esclusione

- le operazioni con le quali i processi accedono a una risorsa comune (*sezioni critiche*) devono escludersi mutuamente nel tempo
- nessun vincolo sull'ordine con il quale le sezioni critiche vengono eseguite
- **Esempio:** A, B sezioni critiche



Sincronizzazione tra processi in ambiente globale: Problema della mutua esclusione

Esempio: P_1 e P_2 interagiscono attraverso una pila condivisa

1) Comportamento corretto

Se sezioni critiche eseguite in mutua esclusione

P_1

.....

```
Top++  
Stack[top]=y
```

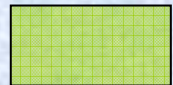
.....

P_2

.....

```
Z=Stack[top]  
Top--
```

.....



sezioni critiche *della stessa classe*

Sincronizzazione tra processi in ambiente globale: Problema della mutua esclusione

Esempio: P_1 e P_2 interagiscono attraverso una pila condivisa

2) Interferenza dipendente dalle velocità di avanzamento reciproche

P_1

.....

Top++

Stack[top]=y

.....

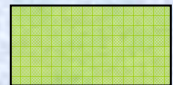
P_2

.....

Z=Stack[top]

Top--

.....



sezioni critiche *della stessa classe*

Sincronizzazione tra processi in ambiente globale:

Problema della mutua esclusione

Soluzione intuitiva, ma errata:

- occupato = 1 ➔ risorsa non disponibile;
- occupato = 0 ➔ risorsa disponibile

P₁

```
prologo: while (occupato==1);  
           occupato =1;  
           <sezione critica A>;  
epilogo: occupato =0;
```

P₂

```
prologo: while (occupato==1);  
           occupato =1;  
           <sezione critica B>;  
epilogo: occupato =0;
```

A, B: sezioni critiche della stessa classe

Sincronizzazione tra processi in ambiente globale:

Problema della mutua esclusione

Soluzione intuitiva: può determinare interferenza:

occupato = 1 → risorsa occupata

occupato = 0 → risorsa libera

P₁

```
prologo: while (occupato==1);  
           occupato =1;  
           <inizia sezione critica A>;  
           <continua sezione critica A>;  
epilogo: occupato =0;
```

P₂

```
prologo: while (occupato==1);  
           occupato =1;  
           <inizia sezione critica B>;  
           <continua sezione critica B>;  
epilogo: occupato =0;
```

Interferenza!

Sincronizzazione tra processi in ambiente globale:

Problema della mutua esclusione

Soluzione intuitiva, ma errata: natura dell'interferenza:

T0: P1 esegue *while* e trova *occupato*=0

T1: P2 esegue *while* e trova *occupato*=0

T2: P1 pone *occupato*=1 ed entra in A

T3: P2 pone *occupato*=1 ed entra in B

➔ Entrambi i processi sono contemporaneamente nella loro sezione critica

➔ Errore dipendente dal tempo

Sincronizzazione tra processi in ambiente globale: Problema della mutua esclusione

Soluzione elementare corretta con istruzione TSL (Test and Set Lock), o istruzioni analoghe

- **TSL R, x:**
 - Il valore contenuto all'indirizzo di memoria x viene copiato nel registro R del processore e contemporaneamente viene scritto un valore diverso da zero all'indirizzo x .
 - Le operazioni di lettura e scrittura sono eseguite in modo indivisibile (hardware): si realizza la contemporanea lettura e modifica di una variabile di memoria
- \Rightarrow P1 può leggere e modificare la variabile *occupato* senza possibilità di interferenze da parte di P2, e viceversa

Sincronizzazione tra processi in ambiente globale: Problema della mutua esclusione

Protocolli *lock* e *unlock*

```
lock(x) :  
loop TSL registro, x      (copia x nel registro e pone x≠0)  
    CMP registro, 0       (il valore iniziale di x era 0 ? )  
    JNE loop              (se no, ricomincia il ciclo)  
    RET                   (ritorna al chiamante)
```

```
unlock(x) :  
    MOV x, 0              (scrive 0 in x)  
    RET                   (ritorna al chiamante)
```

Sincronizzazione tra processi in ambiente globale: Problema della mutua esclusione

Realizzazione della mutua esclusione con protocolli *lock* e *unlock*

P_1

prologo: `lock(x);`
 `<sez. critica A>;`
epilogo: `unlock(x);`

P_2

prologo: `lock(x);`
 `<sez. critica B>;`
epilogo: `unlock(x);`

Sincronizzazione tra processi in ambiente globale: Problema della mutua esclusione

Caratteristiche della soluzione con *lock/unlock*

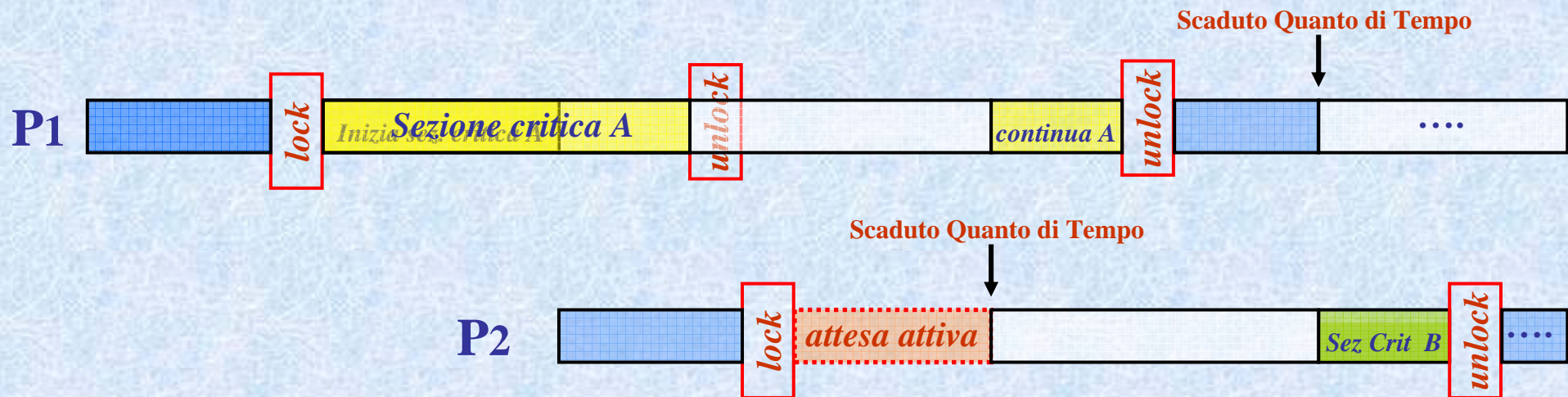
Protocolli *lock* e *unlock* **eseguiti dai processi**, senza intervento del sistema operativo

==> Protocollo *lock* determina attesa attiva per il processo che lo esegue (*busy form of waiting*)

Soluzione accettabile *solo nei multiprocessori e per sezioni critiche brevi*

Sincronizzazione tra processi in ambiente globale: Problema della mutua esclusione

Comportamento di *lock/unlock* nei sistemi uniprocessore



Sincronizzazione tra processi in ambiente globale: Semafori

Semaforo *s*: oggetto del nucleo, associato a un evento.

Attributi:

- **s.value**: intero non negativo, con valore iniziale $s \geq 0$
- **s.queue**: coda di descrittori di processo

==> nella coda sono inseriti i descrittori dei processi che attendono l'evento

Metodi:

- **wait(s)**
- **Signal(s)**

==> uniche operazioni eseguibili dai processi

==> *primitive* invocate con chiamate di sistema



Sincronizzazione tra processi in ambiente globale: Semafori

wait(s)

%% primitiva: funzione del nucleo attivata con SVC; interruzioni disabilitate%

```
{   if (s.value== 0)
    <sospende il processo e inserisce il suo
    descrittore in s.queue;
    interviene lo scheduler per riassegnare il
    processore>;
    else
        s.value= s.value-1;
}
```

%% la primitiva termina con l'istruzione IRET, che riabilita le interruzioni %%

Sincronizzazione tra processi in ambiente globale: Semafori

signal(s)

%% primitiva: funzione del nucleo attivata con SVC; interruzioni disabilitate%%

```
{if ( <esiste almeno un processo in s.queue>)  
    <estrae il primo descrittore in  
    s.queue; modifica in pronto lo stato di  
    questo processo; se previsto preemptive  
    scheduling, interviene lo scheduler per  
    riassegnare(eventualmente)il processore>;  
else  
    s.value = s.value + 1;  
}
```

%% la primitiva termina con l'istruzione IRET, che riabilita le interruzioni %%

Sincronizzazione tra processi in ambiente globale: Semafori

- *wait* e *signal* sono **operazioni atomiche**:
modifica del valore del semaforo ed eventuale sospensione o riattivazione di un processo devono avvenire in modo indivisibile
- Indivisibilità di *wait* e *signal*:
 - processi eseguiti sullo stesso processore: disabilitazione delle interruzioni durante l'esecuzione delle primitive
==> avviene automaticamente grazie al meccanismo della system call.
 - processi eseguiti su processori diversi: disabilitazione delle interruzioni combinata con *lock(x)*, *unlock(x)*.

Sincronizzazione tra processi in ambiente globale: Semafori

Realizzazione di wait e signal per multiprocessore

wait(s)

%% primitiva: funzione del nucleo attivata con SVC; interruzioni disabilitate %%

```
{lock (knucleo)
    if (s.value== 0)
        <sospende il processo e inserisce il suo
        descrittore in s.queue;
        interviene lo scheduler per riassegnare il
        processore>;
    else
        s.value= s.value-1;
unlock (knucleo)
}
```

%% la primitiva termina con l'istruzione IRET, che riabilita le interruzioni %%

Sincronizzazione tra processi in ambiente globale: Semafori

Realizzazione di wait e signal per multiprocessore

signal(s)

%% primitiva: funzione del nucleo attivata con SVC; interruzioni disabilitate%%

```
{lock (knucleo)
    if ( <esiste almeno un processo in s.queue>)
        <estrae il primo descrittore in
        s.queue; modifica in pronto lo stato di
        questo processo; se previsto preemptive
        scheduling, interviene lo scheduler per
        riassegnare(eventualmente)il processore>;
    else
        s.value = s.value + 1;
unlock (knucleo)
}
```

%% la primitiva termina con l'istruzione IRET, che riabilita le interruzioni %%

Sincronizzazione tra processi in ambiente globale: Semafori

Applicazione dei semafori alla soluzione del problema della mutua esclusione

- Semaforo `mutex` associato alla risorsa condivisa
- valore iniziale: `mutex=1`

P_1

```
prologo:   wait(mutex);  
           <sez. critica A>;  
epilogo:  signal(mutex);
```

P_2

```
prologo:   wait(mutex);  
           <sez. critica B>;  
epilogo:  signal(mutex);
```

Sezioni critiche della stessa classe

Processi cooperanti in ambiente globale: problema produttore - consumatore

- Esempio:

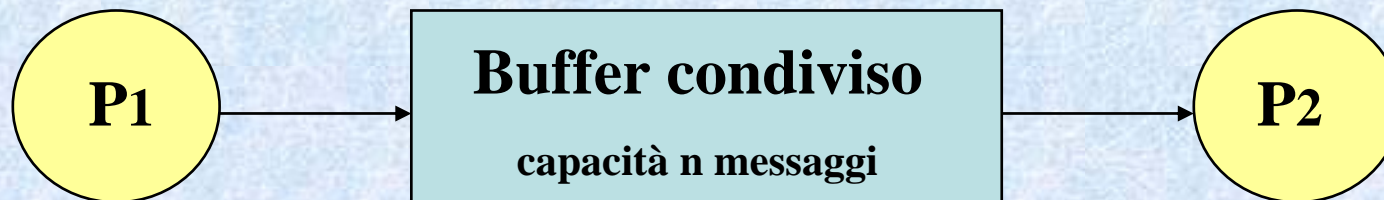
Processo P1 (*produttore*) produce *messaggi* e li invia a P2

Processo P2 (*consumatore*) riceve *messaggi* da P1 e li consuma

Ipotesi:

Messaggi di lunghezza fissa, trasferiti attraverso un buffer condiviso

Capacità del buffer: n messaggi



Processi cooperanti in ambiente globale:

soluzione con semafori del problema produttore - consumatore

Ipotesi: buffer con capacità di 1 messaggio, inizialmente vuoto

Si usano 2 semafori:

- `buffer_disponibile` (valore iniziale 1)
- `messaggio_disponibile` (valore iniziale 0)

Processo Produttore:

```
do {  
    <produce messaggio>;  
    wait (buffer_ disponibile);  
    <deposita nel buffer>;  
    signal (messaggio_ disponibile)  
}  
while (!fine);  
}
```

Processo Consumatore:

```
do {  
    wait (messaggio_ disponibile);  
    <preleva dal buffer>;  
    signal (buffer_ disponibile);  
    <consuma il messaggio>;  
}  
while (!fine);
```

Processi cooperanti in ambiente globale: soluzione con semafori del problema produttore - consumatore

Ipotesi: buffer con capacità di n messaggi, inizialmente vuoto

Si usano i seguenti semafori:

- spazio_disponibile (v.i. n); - messaggio_disponibile (v.i 0)
- mutex (v.i. 1);

Processo Produttore:

```
do {  
    <produce messaggio>;  
    wait (spazio_disponibile);  
    wait (mutex)  
    messaggi_giacenti ++  
    <deposita nel buffer>;  
    signal (mutex)  
    signal (messaggio_disponibile)  
}  
while (!fine);  
}
```

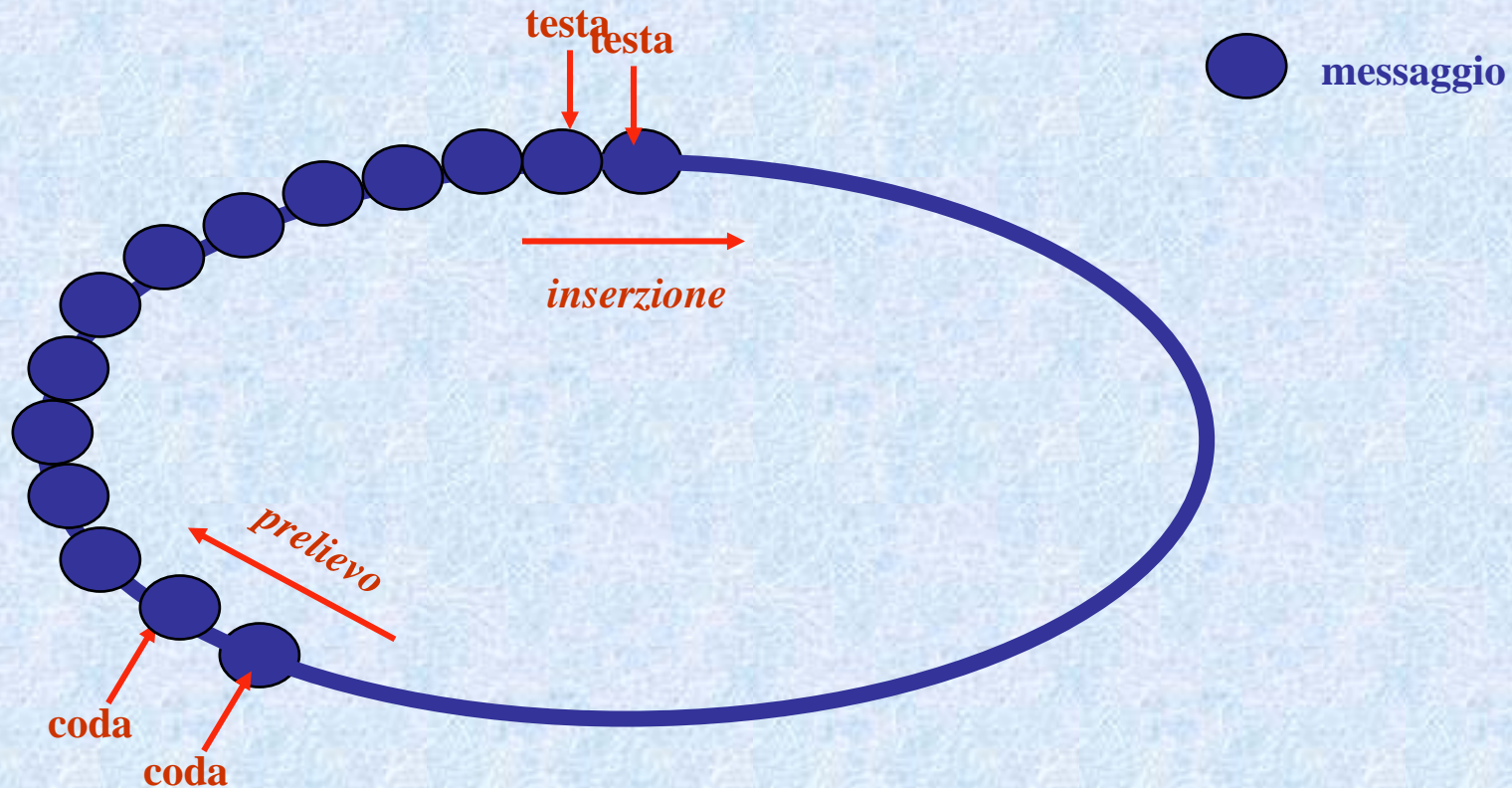
Processo Consumatore:

```
do {  
    wait (messaggio_disponibile);  
    wait (mutex)  
    messaggi_giacenti --  
    <preleva dal buffer>;  
    signal (mutex)  
    signal (spazio_disponibile);  
    <consuma il messaggio>;  
}  
while (!fine);  
}
```

Processi cooperanti in ambiente globale

Problema produttore-consumatore con buffer circolare

- Scambio di messaggi attraverso buffer circolare
- Più produttori, più consumatori ==> **necessaria mutua esclusione sul buffer**



Processi cooperanti in ambiente globale

Problema produttore-consumatore con buffer circolare

Soluzione con semafori

Ipotesi: più produttori, più consumatori

Semafori: *spazio_disponibile* (v.i. N); *messaggio_disponibile* (v.i. 0);

MutexTesta (v.i. 1); *MutexCoda* (v.i. 1) (per mutua esclusione sul buffer tra produttori o consumatori)

Produttore

```
{
  do {
    <produzione del messaggio>
    wait (spazio_disponibile);
    wait(MutexTesta);
    testa=(testa+1)%N;
    <deposito in vettore(testa)>
    signal(MutexTesta);
    signal (messaggio_disponibile)
  }
  while (!fine);
}
```

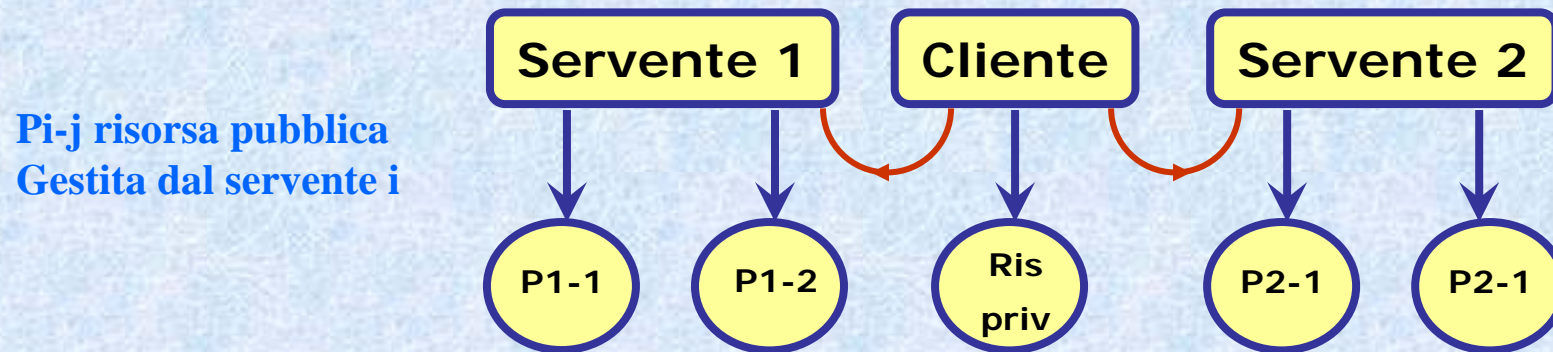
Consumatore

```
{
  do {
    wait (messaggio_disponibile);
    wait(MutexCoda);
    coda=(coda+1)%N
    <prelievo da vettore(coda)>
    signal(MutexCoda);
    signal (spazio_disponibile);
    <consumo del messaggio>
  }
  while (!fine);
}
```

Sincronizzazione tra processi in ambiente locale

Competizione per l'accesso esclusivo alle risorse comuni

- problema analogo a quello della mutua esclusione nell'ambiente globale
- risolto attraverso la cooperazione con i processi serventi



Cooperazione (sincronizzazione diretta o esplicita)

- esempio: per lo scambio di informazione realizzata con meccanismi di comunicazione

Sincronizzazione tra processi in ambiente locale

Comunicazione

Paradigma tipico dei sistemi ad ambiente locale

Canale di comunicazione: oggetto del nucleo

Attributi:

- Buffer, coda di messaggi

==> diverse realizzazioni

Metodi (sono possibili diverse convenzioni per i parametri):

- *send(destinazione, messaggio)*
 - deposita un messaggio nel canale (*messaggio: corpo, mittente, destinatario*)
- *receive(origine, &messaggio)*
 - preleva un messaggio dal canale (*considerando mittente e destinatario*)

==> sono le sole operazioni sui messaggi eseguibili dai processi

==> *primitive* invocate con chiamate di sistema

Sincronizzazione tra processi in ambiente locale

Formato del messaggio



Sincronizzazione tra processi in ambiente locale

Modelli di comunicazione

1) Designazione di destinazione e origine

- diretta
 - *send(dest, &messaggio); receive (mitt, &messaggio)*
- indiretta
 - *send(mailbox, &messaggio); receive (mailbox, &messaggio)*
==> mailbox o porta
- ricezione senza designazione del mittente
 - *receive (&id, &messaggio)*
riceve un messaggio giacente nel canale e restituisce l'identificatore del processo mittente
==> tipica dei processi serventi (interazione "client-server")

Sincronizzazione tra processi in ambiente locale

Modelli di comunicazione

2) Sincronizzazione tra processi comunicanti

- *send* sincrona
 - ==> blocca il mittente se il destinatario non è in attesa di ricevere
 - ==> *combinata con la receive bloccante, realizza il rendez-vous*
- *send* asincrona
 - ==> deposita il messaggio nel canale; non blocca mai il mittente
 - =>> capacità del canale
- *receive* bloccante
 - ==> è la realizzazione normale
- *receive* non bloccante
 - *Protocollo rendez-vous esteso*
 - *Chiamata di procedura remota*

Sincronizzazione tra processi in ambiente locale

Protocollo “Rendez-Vous esteso”

Permette la sincronizzazione tra processi che comunicano con send asincrona e receive bloccante



Processo mittente

.....

Send (destinatario, &mess)

(send asincrona)

Receive(destinatario, &risp)

(receive bloccante)

.....

Processo destinatario

.....

Receive(mittente, &mess)

(receive bloccante)

Send(mittente, &risp)

(send asincrona)

.....

Sincronizzazione tra processi in ambiente locale

Problema produttore-consumatore risolto con meccanismi di comunicazione
==> per diverse specifiche del problema, ricorrere a diversi modelli di comunicazione

Esempio: 1 produttore, 1 consumatore, comunicazione diretta



Processo produttore

```
{
    do {
        produce(&mess) ;
        send(consumatore, &mess) ;
    }
    while(!fine);
}
```

Processo consumatore

```
{
    do {
        receive(produttore, &mess) ;
        consuma(mess) ;
    }
    while(!fine);
}
```

Sincronizzazione tra processi in ambiente locale

Problema produttore-consumatore risolto con meccanismi di comunicazione
==> per diverse specifiche del problema, ricorrere a diversi modelli di comunicazione

Esempio: n produttori, m consumatori, comunicazione indiretta



Processo produttore

```
{  
    do {  
        produce(&mess);  
        send(mailbox, &mess);  
    }  
    while(!fine);  
}
```

Processo consumatore

```
{  
    do {  
        receive(mailbox, &mess);  
        consuma(mess);  
    }  
    while(!fine);  
}
```

Sincronizzazione tra thread

Modelli di interazione:

sono gli stessi caratteristici dei processi in ambiente globale

==> interazione limitata ai thread di uno stesso processo

Ma: meccanismi diversi a seconda della realizzazione dei thread a livello utente o a livello del nucleo

Competizione: un solo thread alla volta deve avere accesso ai dati comuni

==> *Mutua esclusione*

Cooperazione: le operazioni dei thread cooperanti (esempio: problema produttore e consumatore) devono seguire una sequenza corretta

==> *Sincronizzazione diretta o esplicita*

Sincronizzazione tra thread: thread a livello utente

Utilizzabili solo meccanismi a livello utente, che non attivano il nucleo del S.O. (*funzioni della thread library*)

Problema della mutua esclusione: risolto con *lock, unlock*

- lock modificata per gestire lo scheduling dei thread

P₁

prologo: `lock(x);`
 `<sez. critica A>;`
epilogo: `unlock(x);`

P₂

prologo: `lock(x);`
 `<sez. critica B>;`
epilogo: `unlock(x);`

Sincronizzazione tra thread: thread a livello utente

Problema produttore- consumatore risolto con *lock, unlock*

Ipotesi: buffer con capacità di 1 messaggio, inizialmente vuoto

Si usano 2 chiavi binarie:

- `buffer_vuoto` (valore iniziale 1)
- `buffer_pieno` (valore iniziale 0)

==> questa soluzione non può essere estesa banalmente al caso di buffer con capacità N!

Thread Produttore:

```
do {  
    <produce messaggio>;  
    lock (buffer_vuoto);  
    <deposita nel buffer>;  
    unlock (buffer_pieno)  
}  
while (!fine);  
}
```

Thread Consumatore:

```
do {  
    lock (buffer_pieno);  
    <preleva dal buffer>;  
    unlock (buffer_vuoto);  
    <consuma il messaggio>;  
}  
while (!fine);
```

Sincronizzazione tra thread: thread a livello utente

Caratteristica dei protocolli *lock* e *unlock*: *attesa attiva*

Realizzazione della *lock* che riduce la durata dell'attesa attiva:

```
lock(x):  
loop    TSL registro,x           (copia x nel registro e assegna x≠0)  
        CMP registro,0          (il valore iniziale di x era 0 ? )  
        JE fine                 (x=0: fine del protocollo)  
        CALL thread_yield       (x≠0: rilascia il processore)  
        JMP loop                (quando torna in esecuzione, ripete il ciclo)  
fine          RET  
  
unlock(x):  
        MOV x, 0                (scrive 0 in x)  
        RET                     (fine del protocollo)
```

Sincronizzazione tra thread: thread a livello del nucleo

**Sono pienamente utilizzabili i meccanismi del nucleo
(semafori)**

- **Problema della mutua esclusione: risolto con semafori**
- **Problemi di sincronizzazione diretta: risolti con semafori**

**Esempio: problema produttore-consumatore in
qualsiasi formulazione**

Lo standard POSIX per i thread

UNIX, nella versione nativa, non prevede multithreading e non dispone di chiamate di sistema per i threads

- in particolare: *fork* genera *processi* con spazi di indirizzamento disgiunti
- i thread sono realizzabili a livello utente

LINUX è un sistema con multithreading e dispone di chiamate di sistema per i threads

- in particolare chiamata *clone*:
 - genera un thread figlio che condivide (in parte) lo spazio di indirizzamento del padre
 - se lo spazio condiviso è vuoto, genera un processo
- ogni processo ha un thread iniziale che può generare altri thread
- l'unità di schedulazione è il thread

Programmi che utilizzano le chiamate per i thread, e in particolare la *clone*, non sono portabili su UNIX

Per garantire la portabilità, POSIX definisce un insieme di funzioni standard per i thread

- realizzabili con librerie (*livello delle librerie standard*)
- differenti realizzazioni per UNIX e LINUX

Libreria p-thread: sincronizzazione

Mutex: meccanismo per la mutua esclusione nello standard POSIX 1003.1c:

- dati del tipo **pthread_mutex_t**
- sono semafori binari (valore 0 --> *occupato*, valore 1 --> *libero*);
- operazioni fondamentali:
 - inizializzazione: **pthread_mutex_init**
 - locking: **pthread_mutex_lock**
 - unlocking: **pthread_mutex_unlock**

Libreria p-thread: inizializzazione di un *mutex*

L'inizializzazione di un **mutex** si può realizzare con:

```
int pthread_mutex_init(pthread_mutex_t *mutex, const
pthread_mutexattr_t *attr)
```

dove:

- **mutex**: individua il mutex da inizializzare
- **attr**: punta a una struttura che contiene gli attributi del mutex; se NULL, il mutex viene inizializzato a *libero*.

attribuisce un valore iniziale all'intero associato al semaforo (default: *libero*):

Libreria p-thread: operazioni *lock/unlock*

```
int pthread_mutex_lock(pthread_mutex_t *mux);  
int pthread_mutex_unlock(pthread_mutex_t *mux);
```

- **lock**: se il mutex *mux* è occupato, il thread chiamante si sospende; altrimenti occupa il mutex.
- **unlock**: se vi sono thread in attesa del mutex *mux*, ne risveglia uno; altrimenti libera il mutex.

Libreria p-thread: : variabili *condition*

condition: meccanismo per la sincronizzazione nello standard POSIX 1003.1c:

dato del tipo **pthread_cond_t**

- analogamente al semaforo ha una coda sulla quale si sospendono i thread
- a differenza dei semafori, non ha un valore che condiziona la sospensione; sono associate operazioni per la sospensione e la riattivazione *incondizionata* dei thread

==> ma: la sospensione può essere condizionata dal valore di un'espressione calcolato esplicitamente dal thread nel proprio ambiente.

Libreria p-thread: inizializzazione di una *condition*

L'inizializzazione di una *condition* si realizza con:

```
int pthread_cond_init(pthread_cond_t *cond,  
pthread_cond_attr_t *cond_attr)
```

dove:

- **cond** : individua la condizione da inizializzare
- **attr** : punta a una struttura che contiene gli *attributi* della condizione; se NULL, viene inizializzata a default.

Libreria p-thread: uso delle *condition*

Per l'utilizzo delle variabili *condition* sono disponibili le operazioni:

```
pthread_cond_wait(pthread_cond_t *cond,  
                  pthread_mutex_t *mux)
```

- Sospende incondizionatamente il thread sulla *condition* **cond**
- Si può realizzare la sospensione condizionata, condizionando l'esecuzione all'esito della preventiva verifica di una *condizione di accesso*.

```
pthread_cond_signal(pthread_cond_t *cond)
```

- Se esistono thread sospesi sulla *condition*, riattiva il primo; altrimenti non ha effetto

Libreria p-thread: operazione *wait*

Il meccanismo `pthread_cond_wait` può essere combinato con la preventiva verifica di una *condizione di accesso* per realizzare la sospensione condizionata

La verifica dipende generalmente da variabili comuni e pertanto deve essere eseguita in mutua esclusione

==>> necessario associare un *mutex* alla variabile *condition*

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mux)
```

dove:

- **cond**: è la variabile *condition*
- **mux**: è il *mutex* associato

Effetto:

il thread chiamante si sospende sulla coda associata a **cond**, *rilasciando automaticamente il mutex mux* alla successiva riattivazione, il thread *riacquisisce automaticamente mux*.

Libreria p-thread: operazione *signal*

Riattivazione di un thread che si era sospeso sulla *condition* **cond** associata al *mutex* **mutex**:

```
int pthread_cond_signal(pthread_cond_t *cond)
```

==> dopo la `pthread_cond_signal` **deve** essere eseguita la `pthread_cond_unlock`

Effetto:

- a) se esistono thread sospesi nella coda associata a `pthread_cond_signal`, viene selezionato il primo;
- b) se non esistono thread sospesi nella coda associata a `pthread_cond_signal` l'esecuzione di `pthread_cond_unlock` determina il rilascio di `mutex`.

Esempio: uso combinato di condition e mutex

Problema: risorsa utilizzabile da più thread, fino a un massimo di max (1)

```
(max_utenti: condition, mux: mutex)
/*Fase di Entrata: necessaria la mutua esclusione*/
pthread_mutex_lock(&mux);
while(utilizzatori==max)
    pthread_cond_wait(&max_utenti,&mux);
utilizzatori++;
pthread_mutex_unlock(&mux);
```

```
<uso della risorsa>
/* permesso a non più di max utenti senza mutua esclusione */
```

```
/*Fase di Uscita: necessaria la mutua esclusione*/
pthread_mutex_lock(&mux);
utilizzatori--;
pthread_cond_signal(&max_utenti);
pthread_mutex_unlock(&mux);
```